hw4-dl

June 7, 2023

HOMEWORK 4

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms
import numpy as np
import matplotlib.pyplot as plt
from torchvision.datasets import MNIST
```

Shallow Neural Net Classifier

[2]:
```
class ShallowNet(nn.Module):
    def __init__(self, k, p):
        super(ShallowNet, self).__init__()
        self.fc1 = nn.Linear(784, k)
        self.dropout = nn.Dropout(1-p)
        self.fc2 = nn.Linear(k, 10)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return torch.softmax(x, dim=1)
```

[3]:
```
dataset = MNIST(root='./data', download=True, transform=transforms.ToTensor())

data_loader = torch.utils.data.DataLoader(dataset, batch_size=32, shuffle=False)
```

Calculating Mean and STD

[4]:
```
mean = 0.
std = 0.
for images, _ in data_loader:
    batch_samples = images.size(0)
    images = images.view(batch_samples, images.size(1), -1)
    mean += images.mean(2).sum(0)
    std += images.std(2).sum(0)
```

```
mean /= len(data_loader.dataset)
std /= len(data_loader.dataset)

print(f"Calculated mean: {mean}")
print(f"Calculated std: {std}")
```

```
Calculated mean: tensor([0.1307])
Calculated std: tensor([0.3015])
```

[5]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std),
    transforms.Lambda(lambda x: torch.flatten(x))
])
```

[6]:
```
full_train_data = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=transform)
test_data = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=transform)
```

[7]:
```
class_indices = [np.where(np.array(full_train_data.targets) == i)[0][:1000] for␣
 ↪i in range(10)]
subset_indices = np.concatenate(class_indices)
subset_indices = torch.from_numpy(subset_indices)

train_data = Subset(full_train_data, subset_indices)
# Prepare data loaders
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
test_loader = DataLoader(test_data, batch_size=32, shuffle=True)
```

1. Setup your code so that you can run multiple MNIST models for varying choices of k and p automatically, Specifically, you need two for loops (one for k and one for p) and within the loop, you call PyTorch/TensorFlow

[ ]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
K = []
P = []
learning_rate=0.001
results = {}

for k in K:
    for p in P:
        ␣
 ↪print('----------------------------------------------------------------------------------------
        print(f'Running for k={k}, p={p}')
```

```
        # Initialize model
        model = ShallowNet(k, p).to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(),lr=learning_rate)


        train_losses = []
        train_accuracies = []
        test_accuracies = []

        # Training loop
        model.train()
        for epoch in range(80):
            running_loss = 0.0
            correct = 0
            total = 0
            for i, data in enumerate(train_loader, 0):
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward + backward + optimize
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()


                running_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()


            train_losses.append(running_loss / total)
            train_accuracies.append(100 * correct / total)
            train_acc=100 * correct / total
            # Evaluate on test data
            model.eval()
            correct = 0
            total = 0
            with torch.no_grad():
                for data in test_loader:
                    images, labels = data
                    images, labels = images.to(device), labels.to(device)
```

```
                    outputs = model(images)
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()

            test_accuracies.append(100 * correct / total)
            test_acc=100 * correct / total

        print (f' Loss: {running_loss/total}, Train Accuracy: {train_acc} %,␣
↪Test Accuracy: {test_acc} %')
        # Store results
        results[(k, p)] = {
            'train_losses': train_losses,
            'train_accuracies': train_accuracies,
            'test_accuracies': test_accuracies,
        }
```

2) Pick the width grid K = [1, 5, 10, 20, 40] and dropout grid P = [0.1, 0.5, 1.0]. Run MNIST models over these grids with Adam optimizer for 80 epochs. Store the test/train accuracy and loss.

[11]:
```
# Initialize GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

K = [1, 5, 10,20,40]
P = [0.1,0.5,1.0]
learning_rate=0.001
results = {}
for k in K:
    for p in P:
        ␣
↪print('-------------------------------------------------------------------------------------
        print(f'Running for k={k}, p={p}')

        # Initialize model
        model = ShallowNet(k, p).to(device)

        # Define loss function and optimizer
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(),lr=learning_rate)
        train_losses = []
        train_accuracies = []
        test_accuracies = []

        # Training loop
        model.train()
        for epoch in range(80):
```

```
            running_loss = 0.0
            correct = 0
            total = 0
            for i, data in enumerate(train_loader, 0):
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward + backward + optimize
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()

                running_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

            train_losses.append(running_loss / total)
            train_accuracies.append(100 * correct / total)
            train_acc=100 * correct / total
            # Evaluate on test data
            model.eval()
            correct = 0
            total = 0
            with torch.no_grad():
                for data in test_loader:
                    images, labels = data
                    images, labels = images.to(device), labels.to(device)
                    outputs = model(images)
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()

            test_accuracies.append(100 * correct / total)
            test_acc=100 * correct / total
        print (f' Loss: {running_loss/total}, Train Accuracy: {train_acc} %,␣
↪Test Accuracy: {test_acc} %')
        # Store results
        results[(k, p)] = {
            'train_losses': train_losses,
            'train_accuracies': train_accuracies,
            'test_accuracies': test_accuracies,
        }
```

```
--------------------------------------------------------------------------------
----
Running for k=1, p=0.1
 Loss: 0.06702225207090377, Train Accuracy: 28.75 %, Test Accuracy: 26.79 %
--------------------------------------------------------------------------------
----
Running for k=1, p=0.5
 Loss: 0.06730657365322112, Train Accuracy: 28.05 %, Test Accuracy: 27.96 %
--------------------------------------------------------------------------------
----
Running for k=1, p=1.0
 Loss: 0.06589780753850936, Train Accuracy: 35.18 %, Test Accuracy: 35.21 %
--------------------------------------------------------------------------------
----
Running for k=5, p=0.1
 Loss: 0.050369775223732, Train Accuracy: 85.43 %, Test Accuracy: 79.73 %
--------------------------------------------------------------------------------
----
Running for k=5, p=0.5
 Loss: 0.04864052385091781, Train Accuracy: 91.43 %, Test Accuracy: 84.25 %
--------------------------------------------------------------------------------
----
Running for k=5, p=1.0
 Loss: 0.04798439751863479, Train Accuracy: 93.22 %, Test Accuracy: 86.89 %
--------------------------------------------------------------------------------
----
Running for k=10, p=0.1
 Loss: 0.04736659337282181, Train Accuracy: 95.04 %, Test Accuracy: 88.3 %
--------------------------------------------------------------------------------
----
Running for k=10, p=0.5
 Loss: 0.046887760043144225, Train Accuracy: 96.46 %, Test Accuracy: 91.17 %
--------------------------------------------------------------------------------
----
Running for k=10, p=1.0
 Loss: 0.046910217523574826, Train Accuracy: 96.34 %, Test Accuracy: 90.96 %
--------------------------------------------------------------------------------
----
Running for k=20, p=0.1
 Loss: 0.04636979432106018, Train Accuracy: 97.97 %, Test Accuracy: 93.01 %
--------------------------------------------------------------------------------
----
Running for k=20, p=0.5
 Loss: 0.04628583484888077, Train Accuracy: 98.23 %, Test Accuracy: 93.47 %
--------------------------------------------------------------------------------
----
Running for k=20, p=1.0
 Loss: 0.04645389814376831, Train Accuracy: 97.69 %, Test Accuracy: 92.45 %
```

```
--------------------------------------------------------------------------------
----
Running for k=40, p=0.1
 Loss: 0.04614367840290069, Train Accuracy: 98.71 %, Test Accuracy: 94.6 %
--------------------------------------------------------------------------------
----
Running for k=40, p=0.5
 Loss: 0.04612968157529831, Train Accuracy: 98.76 %, Test Accuracy: 94.46 %
--------------------------------------------------------------------------------
----
Running for k=40, p=1.0
 Loss: 0.04614700492620468, Train Accuracy: 98.68 %, Test Accuracy: 94.33 %
```

2a) Fix p = 1.0 which is the case of "no dropout regularization". Plot the test and training accuracy as a function of k. As k increases, does the performance improve? At what k, training accuracy becomes 100%?

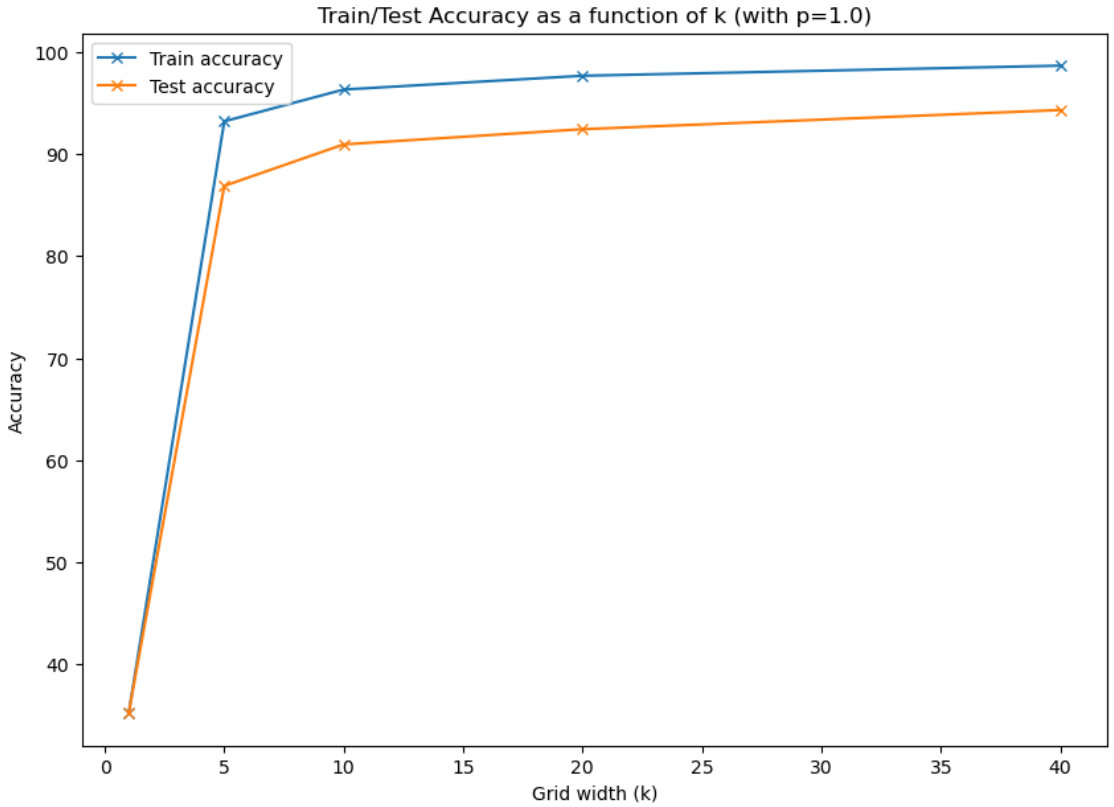
[12]:
```
p = 1.0
train_accur = [results[(k, p)]['train_accuracies'][-1] for k in K]
print(train_accur)
test_accur = [results[(k, p)]['test_accuracies'][-1] for k in K]
print(test_accur)
plt.figure(figsize=(10, 7))

plt.plot(K, train_accur,'-x' ,label='Train accuracy')
plt.plot(K, test_accur, '-x',label='Test accuracy')

plt.title('Train/Test Accuracy as a function of k (with p=1.0)')
plt.xlabel('Grid width (k)')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

```
[35.18, 93.22, 96.34, 97.69, 98.68]
[35.21, 86.89, 90.96, 92.45, 94.33]
```

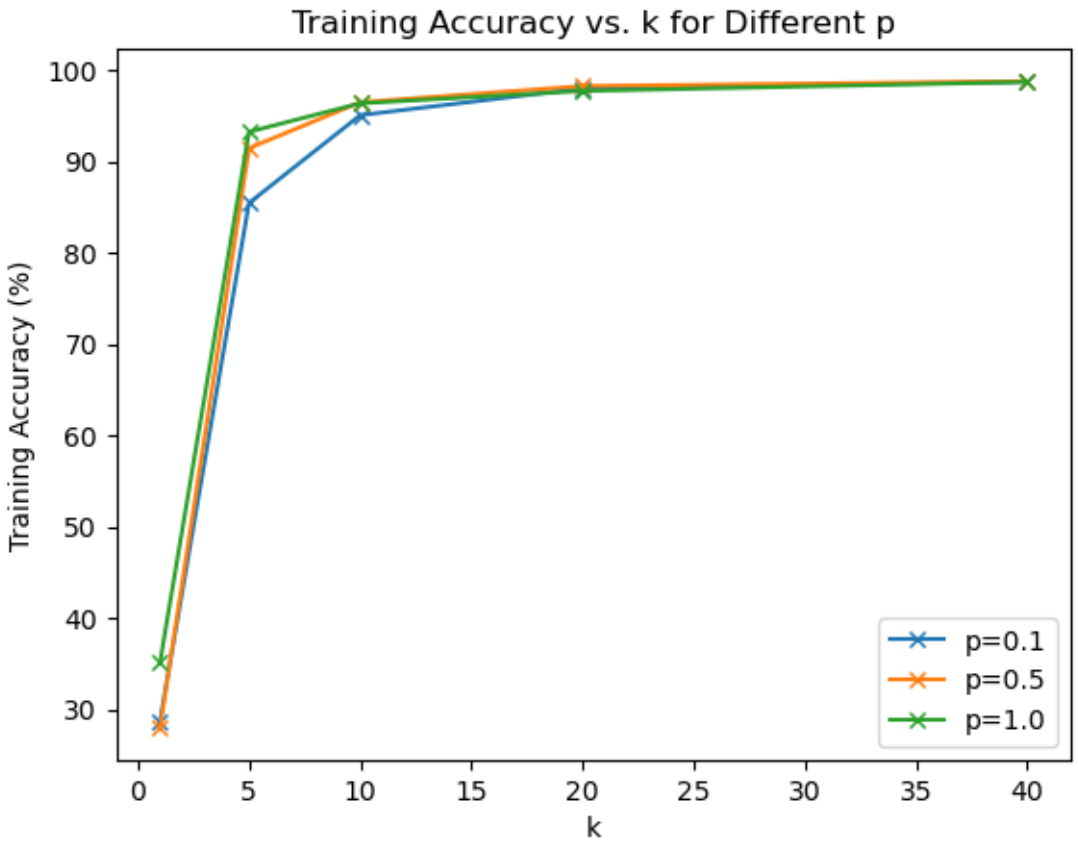


Q)As k increases, does the performance improve? A)As the value of k increases, the accuracy of my model is also increasing. This observation indicates that increasing the value of k has a positive impact on the performance of model. Q)At what k, training accuracy becomes 100%? A)The highest accuracy achieved in my model is 98.76%. The accuracy did not reach 100%.

2b) Plot the training accuracy as a function of k and for different p P on the same plot. What is the role of p on training accuracy? When p is smaller, is it easier to optimize or more difficult? For each choice of p, determine at what choice of k, training accuracy becomes 100%.

[13]:
```
# Plot the training accuracy
for p in P:
    accuracies = [results[(k, p)]['train_accuracies'][-1] for k in K]
    plt.plot(K, accuracies, '-x',label=f'p={p}')

plt.xlabel('k')
plt.ylabel('Training Accuracy (%)')
plt.title('Training Accuracy vs. k for Different p')
plt.legend()
plt.show()
print(accuracies)
```




```
[35.18, 93.22, 96.34, 97.69, 98.68]
```

Q)What is the role of p on training accuracy? A)When the values of p is increasing then the training accuracy is also increasing. Q)When p is smaller, is it easier to optimize or more difficult? A)When the value of "p" in dropout is smaller, it generally makes the optimization process more difficult.

Q)For each choice of p, determine at what choice of k, training accuracy becomes 100%. A)The training accuracy for different values of k and p is as follows:

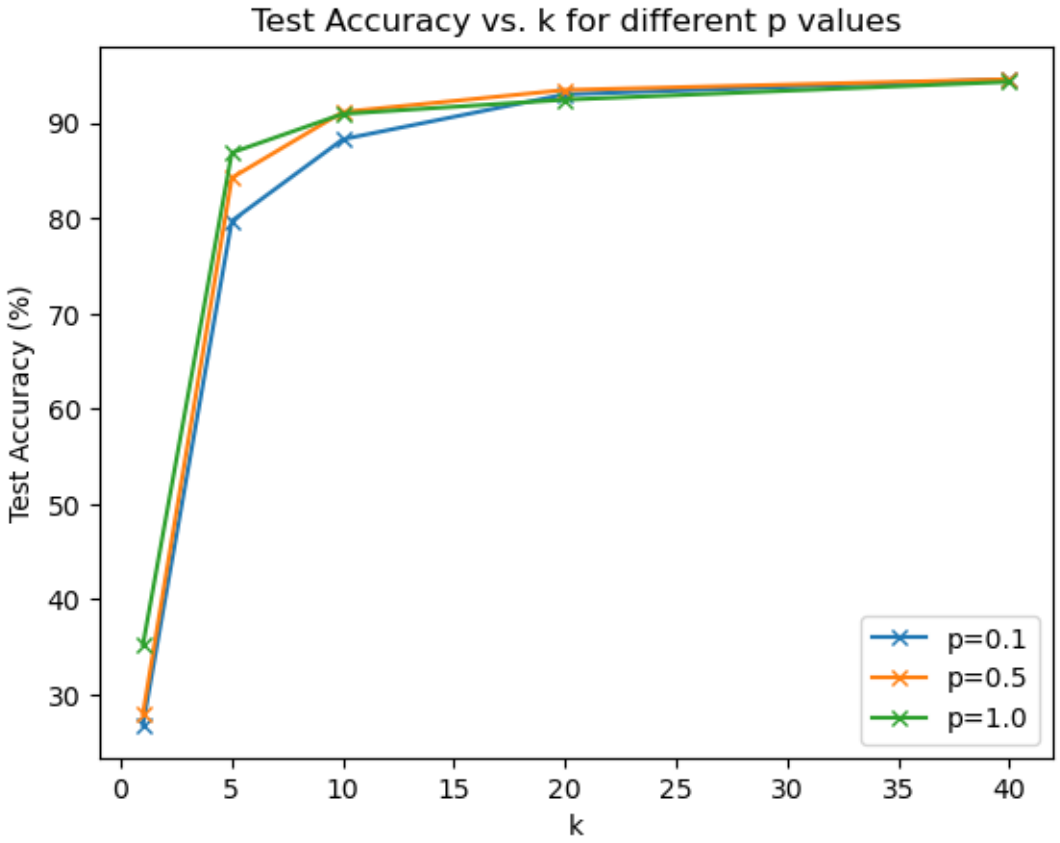
For k=40 and p=0.1, the training accuracy is 98.71%. For k=40 and p=0.5, the training accuracy is 98.76%. For k=40 and p=1.0, the training accuracy is 98.68%. These results indicate that the training accuracy is consistently high but not reaching 100% in any of the cases.

2c) Plot the test accuracy as a function of k and for different p P on the same plot. Does dropout help with the test accuracy? For which (k, p) configuration do you achieve the best test accuracy?

[21]:
```
# Plot test accuracy for different p values
for p in P:
    accuracies = [results[(k, p)]['test_accuracies'][-1] for k in K]
    plt.plot(K, accuracies,'-x', label=f'p = {p}')
```

```
plt.xlabel('k')
plt.ylabel('Test Accuracy (%)')
plt.title('Test Accuracy vs. k for different p values')
plt.legend()
```

[21]: <matplotlib.legend.Legend at 0x7fcd05750e50>



Q)Does dropout help with the test accuracy?A)When the values of p is increasing then the test accuracy is also increasing. Q)For which (k, p) configuration do you achieve the best test accuracy?A)For k=40, p=0.1 the test accuracy is 94.6%.

3. We will spice up the problem by adding some noise to labels. Pick 40% of the training examples at random. Assign their labels at random to another value from 0 to 9. For instance, if the original image is 0 and its label is 0, then you will assign its label to a number from 1 to 9 at random. Thus 60% of the training examples remain correct and 40% will have incorrect labels. Repeat the previous step with this noisy dataset.

[19]:
```
indices = []
for i in range(10):
    class_indices = (full_train_data.targets == i).nonzero().flatten()
    indices.extend(class_indices[:1000].tolist())
```

```
num_to_change = len(indices) * 4 // 10
indices_to_change = np.random.choice(indices, num_to_change, replace=False)
for i in indices_to_change:
    original_label = full_train_data.targets[i].item()
    new_label = np.random.choice([x for x in range(10) if x != original_label])
    full_train_data.targets[i] = new_label

train_data = Subset(full_train_data, indices)


train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
test_loader = DataLoader(test_data, batch_size=32, shuffle=True)
```

[20]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

K = [1, 5,10,20 ,40]
P = [ 0.1,0.5,1.0]
learning_rate=0.001
results = {}
for k in K:
    for p in P:
        print('-----------------------------------------------------')
        print(f'Running for k={k}, p={p}')

        # Initialize model
        model = ShallowNet(k, p).to(device)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(),lr=learning_rate)


        train_losses = []
        train_accuracies = []
        test_accuracies = []

        # Training loop
        model.train()
        for epoch in range(80):
            running_loss = 0.0
            correct = 0
            total = 0
            for i, data in enumerate(train_loader, 0):
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()
```

```
            # forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_losses.append(running_loss / total)
        train_accuracies.append(100 * correct / total)
        train_acc=100 * correct / total

        # Evaluate on test data
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for data in test_loader:
                images, labels = data
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        test_accuracies.append(100 * correct / total)
        test_acc=100 * correct / total

    print (f'Epoch [{epoch+1}/{80}], Loss: {running_loss/total}, Train␣
↪Accuracy: {train_acc} %, Test Accuracy: {test_acc} %')

    results[(k, p)] = {
        'train_losses': train_losses,
        'train_accuracies': train_accuracies,
        'test_accuracies': test_accuracies,
    }
```

```
--------------------------------------------------
Running for k=1, p=0.1
Epoch [80/80], Loss: 0.07103709373474121, Train Accuracy: 14.38 %, Test
Accuracy: 17.22 %
--------------------------------------------------
Running for k=1, p=0.5
```

```
Epoch [80/80], Loss: 0.07044226641654969, Train Accuracy: 17.34 %, Test
Accuracy: 28.16 %
--------------------------------------------------
Running for k=1, p=1.0
Epoch [80/80], Loss: 0.0720646769285202, Train Accuracy: 10.28 %, Test Accuracy:
8.92 %
--------------------------------------------------
Running for k=5, p=0.1
Epoch [80/80], Loss: 0.06576686786413193, Train Accuracy: 36.43 %, Test
Accuracy: 70.46 %
--------------------------------------------------
Running for k=5, p=0.5
Epoch [80/80], Loss: 0.06467315591573715, Train Accuracy: 39.73 %, Test
Accuracy: 78.72 %
--------------------------------------------------
Running for k=5, p=1.0
Epoch [80/80], Loss: 0.0660957652926445, Train Accuracy: 35.46 %, Test Accuracy:
69.51 %
--------------------------------------------------
Running for k=10, p=0.1
Epoch [80/80], Loss: 0.06455691485404969, Train Accuracy: 40.16 %, Test
Accuracy: 77.74 %
--------------------------------------------------
Running for k=10, p=0.5
Epoch [80/80], Loss: 0.06417175824642181, Train Accuracy: 41.18 %, Test
Accuracy: 81.45 %
--------------------------------------------------
Running for k=10, p=1.0
Epoch [80/80], Loss: 0.06378707195520401, Train Accuracy: 42.39 %, Test
Accuracy: 82.46 %
--------------------------------------------------
Running for k=20, p=0.1
Epoch [80/80], Loss: 0.0639454555630684, Train Accuracy: 41.8 %, Test Accuracy:
82.82 %
--------------------------------------------------
Running for k=20, p=0.5
Epoch [80/80], Loss: 0.06359373109340667, Train Accuracy: 42.82 %, Test
Accuracy: 82.82 %
--------------------------------------------------
Running for k=20, p=1.0
Epoch [80/80], Loss: 0.06362250114679337, Train Accuracy: 42.79 %, Test
Accuracy: 83.46 %
--------------------------------------------------
Running for k=40, p=0.1
Epoch [80/80], Loss: 0.06312838342189789, Train Accuracy: 44.28 %, Test
Accuracy: 82.05 %
--------------------------------------------------
Running for k=40, p=0.5
```

```
Epoch [80/80], Loss: 0.06329313769340515, Train Accuracy: 43.84 %, Test
Accuracy: 83.03 %
--------------------------------------------------
Running for k=40, p=1.0
Epoch [80/80], Loss: 0.06343278976678848, Train Accuracy: 43.36 %, Test
Accuracy: 82.92 %
```

3a)Fix p = 1.0 which is the case of "no dropout regularization". Plot the test and training accuracy as a function of k. As k increases, does the performance improve? At what k, training accuracy becomes 100%?

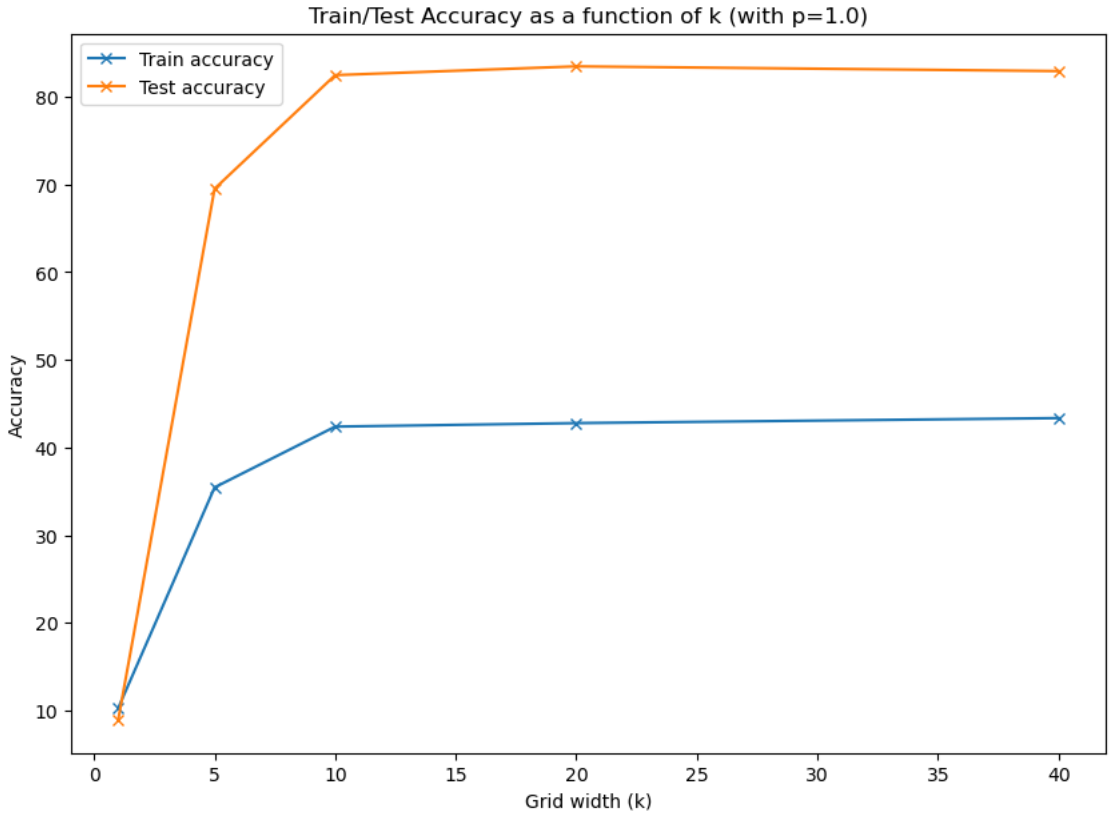
[22]:
```
p = 1.0
train_accur = [results[(k, p)]['train_accuracies'][-1] for k in K]
print(train_accur)
test_accur = [results[(k, p)]['test_accuracies'][-1] for k in K]
print(test_accur)
plt.figure(figsize=(10, 7))

plt.plot(K, train_accur,'-x', label='Train accuracy')
plt.plot(K, test_accur,'-x', label='Test accuracy')

plt.title('Train/Test Accuracy as a function of k (with p=1.0)')
plt.xlabel('Grid width (k)')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

```
[10.28, 35.46, 42.39, 42.79, 43.36]
[8.92, 69.51, 82.46, 83.46, 82.92]
```

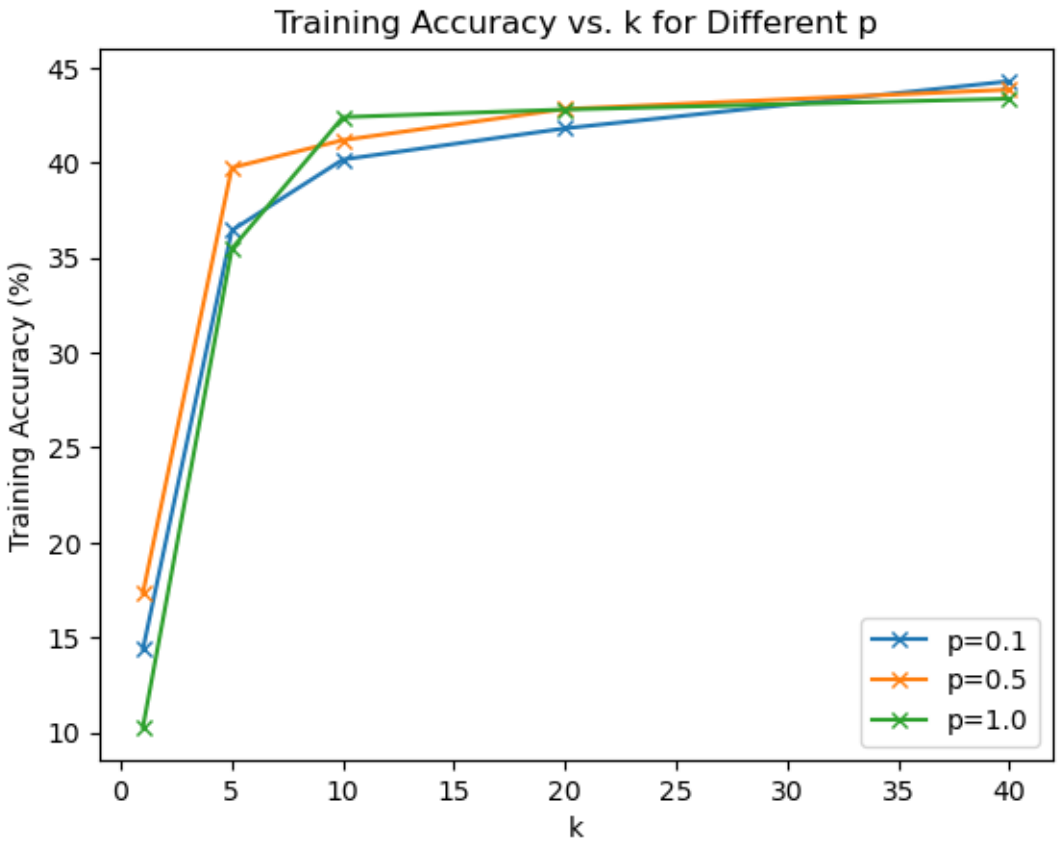


Q)As k increases, does the performance improve? A)As the value of k increases, the accuracy of my model is also increasing. This observation indicates that increasing the value of k has a positive impact on the performance of model. Q)At what k, training accuracy becomes 100%? A)The highest accuracy achieved in my model is 44.28%. The accuracy did not reach 100%,

3b) Plot the training accuracy as a function of k and for different p P on the same plot. What is the role of p on training accuracy? When p is smaller, is it easier to optimize or more difficult? For each choice of p, determine at what choice of k, training accuracy becomes 100%

[23]:
```
# Plot the training accuracy
for p in P:
    accuracies = [results[(k, p)]['train_accuracies'][-1] for k in K]
    plt.plot(K, accuracies,'-x', label=f'p={p}')

plt.xlabel('k')
plt.ylabel('Training Accuracy (%)')
plt.title('Training Accuracy vs. k for Different p')
plt.legend()
plt.show()
```



Q)What is the role of p on training accuracy? A)When the values of p is increasing then the training accuracy is also increasing for most of the cases. Q)When p is smaller, is it easier to optimize or more difficult? A)When the value of "p" in dropout is smaller, it generally makes the optimization process more difficult.

Q)For each choice of p, determine at what choice of k, training accuracy becomes 100%. A)The training accuracy for different values of k and p is as follows:

For k=40 and p=0.1, the training accuracy is 44.28%. For k=40 and p=0.5, the training accuracy is 43.84%. For k=40 and p=1.0, the training accuracy is 43.36%. My training accuracy is not reaching 100% in any of the cases.

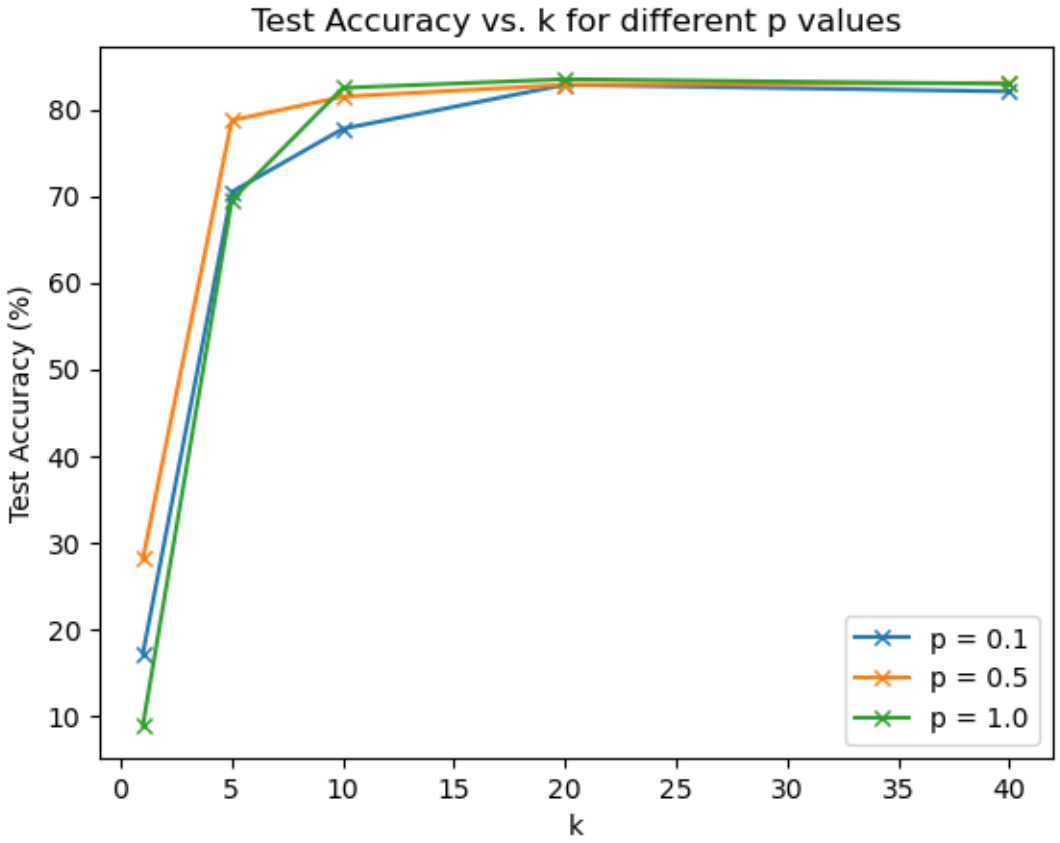
3c) Plot the test accuracy as a function of k and for different p P on the same plot. Does dropout help with the test accuracy? For which (k, p) configuration do you achieve the best test accuracy?

[24]:
```
# Plot test accuracy for different p values
for p in P:
    accuracies = [results[(k, p)]['test_accuracies'][-1] for k in K]
    plt.plot(K, accuracies,'-x', label=f'p = {p}')

plt.xlabel('k')
plt.ylabel('Test Accuracy (%)')
```

```
plt.title('Test Accuracy vs. k for different p values')
plt.legend()

plt.show()
```



Q)Does dropout help with the test accuracy?A)When the values of p is increasing then the test accuracy is also increasing. Q)For which (k, p) configuration do you achieve the best test accuracy?A)For k=20, p=1 the test accuracy is 83.46%.

4) Comment on the differences between Step 2 and Step 3.A)The key difference between Step 2 and Step 3 lies in the introduction of noise to the dataset. In Step 2, no noise is introduced, whereas in Step 3, we deliberately add noise to the dataset. Q)How does noise change things?A)Due to the introduction of noise in Step 3, the training process becomes more challenging, resulting in lower accuracies compared to Step 2.

With the presence of noise in the dataset, the learning task becomes more complex for the model. The noise can introduce additional variations, uncertainties, or mislabeled samples, making it harder for the model to accurately learn and generalize from the data. As a result, the model may struggle to achieve high accuracies during training. Q)For which setup dropout is more useful?A)Dropout regularization helps in dealing with overfitting and can assist the model generalize better as it is pushed to acquire more distributed representations, therefore we can talk about which step benefits

from dropout, step 3 is the one that benefits.Consequently, it enhances its capacity to generalize in the presence of unseen data.